

18 | a + b：动态类型是灾难之源还是最好的特性？（上）

2019-12-25 周爱民

JavaScript核心原理解析

[进入课程 >](#)



讲述：周爱民

时长 20:34 大小 18.84M



你好，我是周爱民，欢迎回到我的专栏。今天我们讲的主题是 JavaScript 的动态类型系统。

动态类型是 JavaScript 的动态语言特性中最有代表性的一种。

动态执行与动态类型是天生根植于 JavaScript 语言核心设计中的基础组件，它们相辅相成，导致了 JavaScript 在学习上是易学难精，在使用中是易用易错。成兹败兹，难以得失论。

类型系统的简化

从根底上来说，JavaScript 有着两套类型系统，如果仅以此论，那么还算不上复杂。

但是 ECMAScript 对语言类型的约定，又与 JavaScript 原生的、最初的语言设计不同，这导致了各种解释纷至沓来，很难统一成一个说法。而且，ECMAScript 又为规范书写而订立了一套类型系统，并不停地演进它。这就如同雪上加霜，导致 JavaScript 的类型系统越发地说不清楚了。

在讨论动态类型的时候，可以将 JavaScript 类型系统做一些简化，从根底里来说，JavaScript 也就是 `typeof()` 所支持的 7 种类型，其中的“**对象 (object)**”与“**函数 (function)**”算一大类，合称为**引用类型**，而其他类型作为**值类型**。

无论如何，我们且先以这种简单的类型划分为基础，来讨论 JavaScript 中的动态类型。因为这样一来，JavaScript 中的类型转换变得很简单、很干净，也很易懂，可以用两条规则概括如下：

1. 从值 `x` 到引用，调用 `Object(x)` 函数。
2. 从引用 `x` 到值，调用 `x.valueOf()` 方法；或调用 4 种值类型的包装类函数，例如 `Number(x)`，或者 `String(x)` 等等。

简单吧？当然不会这么简单。

先搞定一半

在**类型转换**这件事中，有“半件”是比较容易搞定的。

这个一半，就是“**从值 `x` 到引用**”。因为主要的值类型都有对应的引用类型，因此 JavaScript 可以用简单方法——对应地将它们转换过去。

使用 `Object(x)` 来转换是很安全的方法，在用户代码中不需要特别关心其中的 `x` 是什么样的数据——它们可以是特殊值（例如 `null`、`undefined` 等），或是一般的值类型数据，又或者也可以是一个对象。所有使用 `Object(x)` 的转换结果，都将是一个尽可能接近你的预期的**对象**。例如，将数字值转换成数字对象：

```
1 > x = 1234;
```

```
2
3 > Object(x);
4 [Number: 1234]
```

类似的还包括字符串、布尔值、符号等。而 `null`、`undefined` 将被转换为一个一般的、空白的对象，与 `new Object` 或一个空白字面量对象（也就是 `{ }`）的效果一样。这个运算非常好用的地方在于，如果 `x` 已经是一个对象，那么它只会返回原对象，而不会做任何操作。也就是说，它没有任何的副作用，对任何数据的预期效果也都是“返回一个对象”。而且在语法上，`Object(x)` 也类似于一个类型转换运算，表达的是将任意 `x` 转换成对象 `x`。

简单的这“半件事”说完后，我们反过来，接着讨论将**对象转换成值**的情况。

值 VS 原始值 (Primitive values)

任何对象都会有继承自原型的两个方法，称为 `toString()` 和 `valueOf()`，这是 JavaScript 中“对象转换为值”的关键。

一般而言，你可以认为“任何东西都是可以转换为字符串的”，这个很容易理解，比如 `JSON.stringify()` 就利用了这一个简单的假设，它“几乎”可以将 JavaScript 中的任何对象或数据，转换成 JSON 格式的文本。

所以，我的意思是说，在 JavaScript 中将任何东西都转换成字符串这一点，在核心的原理上，以及具体的处理技术上都不存在什么障碍。

但是如何理解“**将函数转换成字符串**”呢？

从最基础的来说，函数有两个层面的含义，一个是它的可执行代码，也就是文本形式的源代码；另一个则是函数作为对象，也有自己的属性。

所以，“理论上来说”，函数也可以被作为一个对象来转换成字符串，或者说，序列化成本形式。

又或者再举一个例子，我们需要如何来理解将一个“符号对象”转换成“符号”呢？是的，我想你一定会说，没有“符号对象”这个东西，因为符号是值，不是对象。其实这样讲只是对了一半，因为现实中确实可以将一个“符号值”转换为一个“符号对象”，例如：

```
1 > x = Object(Symbol())  
2 [Symbol: Symbol()]
```

那么在这种情况下，这个“符号对象 x”又怎么能转换为字符串呢？

所以，“一切都能转换成字符串”只是理论上行得通，而实际上很多情况下是做不到的。

在这些“无法完成转换”的情况下，JavaScript 仍然会尝试给出一个有效的字符串值。基本上，这种转换只能保证“不抛出异常”，而无法完成任何有效的计算。例如，你在通常情况下将对象转换为字符串，就只会得到一个“简单的描述”：

```
1 > (new Object).toString()  
2 '[object Object]'
```

为了将这个问题“一致化”——也就是将问题收纳成更小的问题，JavaScript 约定，所有“对象 -> 值”的转换结果要尽量地趋近于 string、number 和 boolean 三者之一。不过这从来都不是“书面的约定”，而是因为 JavaScript 在早期的作用，就是用于浏览器上的开发，而：

浏览器可以显示的东西，是 string；


可以计算的东西，是 number；

可以表达逻辑的东西，是 boolean。

因此，在一个“最小的、可以被普通人理解的、可计算的程序系统中”，支持的“值类型数据”的最小集合，就应该是这三种。

这个问题不仅仅是浏览器，就算是一台放在云端的主机，你想要去操作它，那么通过控制台登录之后的 shell 脚本，也必须支持它。更远一点地说，你远程操作一台计算机，与浏览器用户要使用 gmail，这二者在计算的抽象上是一样的，只是程序实现的复杂性不一样而已。

所以，对于（ECMAScript 5 以及之前的）JavaScript 来说，当它支持值转换向“对应的”对象时，或者反过来从这些对象转换回值的时候，所需要处理的也无非是这三种类型而已。而处理的具体方法也很简单，就是在使用 `Object(x)` 来转换得到的对象实例中添加一个内部槽，存放这个 `x` 的值。更确切地说，下面两行代码在语义上的效果是一致的：

 复制代码

```
1 obj = Object(x);
2
3 // 等效于（如果能操作内部槽的话）
4 obj.[[PrimitiveValue]] = x;
```

于是，当需要从对象中转换回到值类型时，也就是把这个 `PrimitiveValue` 值取出来就可以了。而“**取出这个值，并返回给用户代码**”的方法，就称为 `valueOf()`。

到了 ECMAScript 6 中，这个过程就稍稍有些不同，这个内部槽是区别值类型的，因此为每种值类型设计了一个独立的私有槽名字。加上 ES8 中出现的大整数类型（`BigInt`），一共就有了 5 个对应的私有槽：`[[BooleanData]]`、`[[NumberData]]`、`[[StringData]]`、`[[SymbolData]]` 和 `[[BigIntData]]`。其中除了 `Symbol` 类型之外，都是满足在上面所说的：

一个“最小的、可以被普通人理解的、可计算的程序系统中”，支持的“值类型数据”的最小集合

这样的一个设定的。

那么“符号”这个东西出现的必要性何在呢？

这个问题我就不解释了，算作本讲的课后习题之一，希望你可以踊跃参与讨论。不过就问题的方向来说，仍然是出于**计算系统的完备性的**。如果你非要说这个是因为张三李四喜欢，某个 tc39 提案者的心头好，这样的答案就算是当事人承认，我也是不认可的。：)

好。回到正题。那么在 ECMAScript 6 之后，除 `[[PrimitiveValue]]` 这个私有槽变成了 5 种值类型对应的、独立的私有槽之外，还有什么不同呢？

是的，这个你可能也已经注意到了。ECMAScript 6 之后还出现了 `Symbol.toPrimitive` 这个符号。而它，正是将原本的 `[[PrimitiveValue]]` 这个私有槽以及其访问过程标准化，然后暴露给 JavaScript 用户编程的一个界面。

说到这里，就必须明确**一般的值** (Values) 与**原始值** (Primitive values) 之间的关系了。

不过，在下一步的讨论之前，我要先帮你总结一下前面的内容：

也就是说，从 `typeof(x)` 的 7 种结果类型来看，其中 `string`、`boolean`、`number`、`bigint` 和 `symbol` 的值类型与对象类型转换，就是将该值存入私有槽，或者从私有槽中把相应的值取出来就好了。

在语言中，这些对应的对象类型被称为“包装类”，与此相关的还有“装箱”与“拆箱”等等行为，这也是后续会涉及到的内容。

NOTE: 在 ECMAScript 6 之前，由于 `[[PrimitiveValue]]` 来存放对应的封装类。也就是说，只有当 `obj.[Class]` 存放着 `false` 值时，它才是 `false` 值所对应的对象实例。

而 ECMAScript 6 将上述的依赖项变成了一个，也就是说只要有一个对象有内部槽 `[[BooleanData]]`，那么它就是某个 `boolean` 值对应的对象。这样处理起来就简便了，不必每次做两项判断。

所以，一种关于“原始值”的简单解释是：所有 5 种能放入私有槽（亦即是说它们有相应的包装类）的值 (Values)，都是原始值；并且，再加上两个特殊值 `undefined` 和 `null`，那么就是所谓原始值 (Primitive values) 的完整集合了。

接下来，如果转换过程发生在“值与值”之间呢？

干掉那两个碍事儿的

`bigint` 这个类型最好说，它跟 `number` 在语言特性上是一回事儿，所以它的转换没什么特殊性，下面在我会在讲到 `number` 的时候，一并讲解。

除此之外，还有两个类型在与其他类型的转换中是简单而特殊的。

例如，**symbol** 这个值类型，它其实既没有办法转换成别的类型，也没有办法从别的类型转换过来。无论是哪种方式转换，它在语义上都是丢失了的、是没有意义的。当然，现实中你也可以这么用，比如：

 复制代码

```
1 > console.log(Symbol())
2 Symbol()
```

这里的确发生了一个 “symbol -> string” 的转换。但它的结果只能表示这是一个符号，至于哪个符号，符号 a 还是符号 b，全都分不出来。类似于此，所有 “符号 -> 其他值类型” 的转换不需要太特别的讨论，由于所有能发生的转换都是定值，所以你可以做一张表格出来对照参考即可。当然，如果是 “其他值类型 -> symbol” 的这种转换，实际结果就是创建一个新符号，而没有 “转换” 的语义了。

另外一个碍事儿的也特别简单，就是 **boolean**。

ECMAScript 为了兼容旧版本的 JavaScript，直接将这个转换定义成了一张表格，这个表格在 ECMAScript 规范或者我们常用的 [MDN](#) (Mozilla Developer Network) 上可以直接查到。简单地说，就是除了 undefined、null、0、NaN、"" (empty string) 以及 BigInt 中的 0n 返回 false 之外，其他的值转换为 boolean 时，都将是 true 值。

当然，不管怎么说，要想记住这些类型转换并不容易（当然也不难），简单地做法，就是直接把它们包装类当作函数来调用，转换一下就好了。在你的代码中也可以这么写，例如：

 复制代码

```
1 > x = 100n; // `bigint` value
2 > String(x) // to `string` value
3 '100n'
4
5 > Boolean(x); // to `boolean` value
6 true
```

这些操作简单易行，也不容易出错，用在代码中还不影响效率，一切都很好。

NOTE: 这些可以直接作为函数调用的包装类，一共有四个，包括 `String()`、`Number()`、`Boolean()` 和 `BigInt()`。此外，`Symbol()` 在形式上与此相同，但执行语义是略有区别的。

但并不那么简单。因为我还没有跟你讨论过字符串和数字值的转换。

以及，还有特别要命的“隐式转换”。

隐式转换

由于函数的参数没有类型声明，所以用户代码可以传入任何类型的值。对于 JavaScript 核心库中的一些方法或操作来说，这表明它们需要一种统一、一致的方法来处理这种类型差异。例如说，要么拒绝“类型不太正确的参数”，抛出异常；要么用一种方式来使这些参数“变得正确”。

后一种方法就是“隐式转换”。但是就这两种方法的选择来说，JavaScript 并没有编码风格层面上的约定。基本上，早期 JavaScript 以既有实现为核心的时候，倾向于让引擎吞掉类型异常（`TypeError`），尽量采用隐式转换来让程序在无异常的情况下运行；而后期，以 ECMAScript 规范为主导的时候，则倾向于抛出这些异常，让用户代码有机会处理类型问题。

隐式转换最主要的问题就是会带来大量的“潜规则”。

例如经典的 `String.prototype.search(r)` 方法，其中的参数从最初设计时就支持在 `r` 参数中传入一个字符串，并且将隐式地调用 `r = new RegExp(r)` 来产生最终被用来搜索的正则表达式。而 `new RegExp(r)` 这个运算中，由于 `RegExp()` 构造器又会隐式地将 `r` 从任何类型转换为字符串类型，因而在这整个过程中，向原始的 `r` 参数传入任何值都不会产生任何的异常。

例如，其实你写出下面这样的代码也是可以运行的：

```
1 > "aa1aa".search(1)
2 2
3
4 > "000false111".search(0 > 5)
```

 复制代码

隐式转换导致的“潜规则”很大程度上增加了理解用户代码的难度，也不利于引擎实现。因此，ECMAScript 在后期就倾向于抛弃这种做法，多数的“新方法”在发现类型不匹配的时候，都设计为显式地抛出类型错误。一个典型的结果就是，在 ECMAScript 3 的时代，TypeError 这个词在规范中出现的次数是 24 次；到了 ECMAScript 5，是 114 次；而 ECMAScript 6 开始就暴增到 419 次。

因此，越是早期的特性，越是更多地采用了带有“潜规则”的隐式转换规则。然而很不幸的是，几乎所有的“运算符”，以及大多数常用的原型方法，都是“早期的特性”。

所以在类型转换方面，JavaScript 成了“潜规则”最多的语言之一。

好玩的

@graybernhardt 曾在 2012 年发布过一个 [演讲](#) (A lightning talk by Gary Bernhardt from CodeMash 2012)，提到一个非常非常著名的案例，来说明这个隐式转换，以及它所带来的“潜规则”有多么的不可预测。这个经典的示例是：

将 [] 和 {} 相加，会发生什么？

尝试一下这个 case，你会看到：

```
1 > [] + {}
2 '[object Object]'
3
4 > {} + []
5 0
6
7 > {} + {}
8 NaN
9
10 > [] + []
11 ''
```

 复制代码

嗯！四种情况居然没有一个是相同的！

不过有一点需要注意到的，就是输出的结果，总是会“收敛”到两种类型：字符串，或者数值。嗯，“隐式转换”其实只是表面现象，核心的问题是，这种转换的结果总是倾向于“string/number”两种值类型。

这个，才是我们这一讲要讲“大问题”。

且听下回分解

到现在为止，这一节课其实才开了个头，也就是对“a + b”这个标题做了一个题解而已。这主要是因为 JavaScript 中有关类型处理的背景信息太多、太复杂，而且还处在不停的变化之中。许多稍早的信息，与现在的应用环境中的现状，或者你手边可备查的资料之间都存在着不可调和的矛盾冲突，因此对这些东西加以梳理还原，实在是大有必要的。这也就是为什么这一讲会说到现在，仍然没有切入正题的原因。

当然，一部分原因也在于：这些絮絮叨叨的东西，也原本就是“正题”的一部分。比如说，你至少应该知道的内容包括：

语言中的引用类型和值类型，以及 ECMAScript 中的原始值类型（Primitive values）之间存在区别；

语言中的所谓“引用类型”，与 ECMAScript 中的“引用（规范类型）”是完全不同的概念；

所有值通过包装类转换成对象时，这个对象会具有一个内部槽，早期它统一称为 `[[PrimitiveValue]]`，而后来 JavaScript 为每种包装类创建了一个专属的；

使用 `typeof(x)` 来检查 x 的数据类型，在 JavaScript 代码中是常用而有效方法；

原则上来说，系统只处理 boolean/string/number 三种值类型（bigint 可以理解为 number 的特殊实现），其中 boolean 与其他值类型的转换是按对照表来处理的。

总地来说，类型在 JavaScript 中的显式转换是比较容易处理的，而标题“a + b”其实包含了太多隐式转换的可能性，因此尤其复杂。关于这些细节，且听下回分解。

这一讲没有复习题。不过如果你愿意，可以把上面讲到的 @graybernhardt 的四个示例尝试一下，解释一下它们为什么是这个结果。

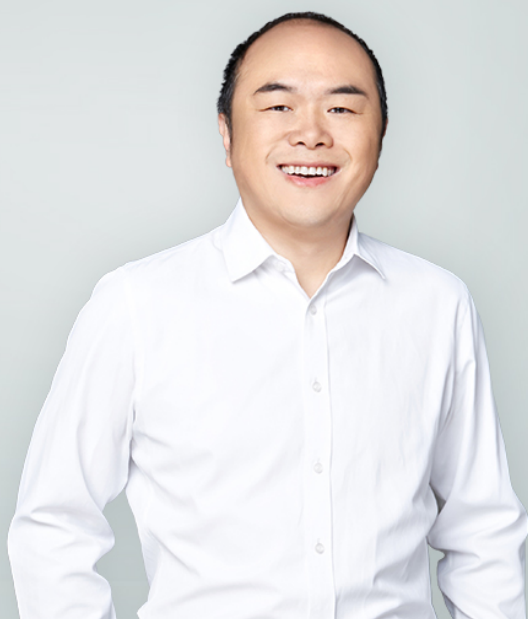
而下一讲，我再来为你公布答案，并且做详细解说。

点击参与 

打卡 46 天，彻底搞定 JavaScript



扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | Object.setPrototypeOf(x, null)：连Brendan Eich都认错，但null值还活着

下一篇 19 | a + b：动态类型是灾难之源还是最好的特性？（下）

精选留言 (7)

 写留言



Y

2019-12-25

这应该这是由于对象类型转换为值类型时的拆箱操作导致的。

[]拆箱的话会先执行[].valueOf(),得到的是[],并不是原始值，就执行[].toString(),得到的结果是"。

{}拆箱会先执行{}.valueOf(),得到的是{},并不是原始值，于是执行toString(),得到的结果是[object Object]。...

展开 ▾

作者回复: 赞的。^^.

今天的课程可以对答案哟。



4



潇潇雨歇

2019-12-25

[]和{}在转换为值类型时会先调用valueOf然后调用toString。

- 1、[]+{}, 前者转换",后者转换为[object Object], 这里为字符串连接操作, 所以结果为'[object Object]'
- 2、{}+[], 前者为代码块, 后者+操作符将"转换为0, 所以结果为0
- 3、{}+{}, 前者为代码块, 后者+操作符将'[object Object]'转换为NaN, 因为它不能转...

展开

作者回复: 赞的, +1票。^^.

今天的课程就分析这个了。



1



sprinty

2019-12-25

在您的文章里, 经常出现 "界面" 这个词, 怎么理解呢?

我简单的当做 "编程接口" 的近义词。

展开

作者回复: 是的。同义。

只是我一直的工作是架构, 所以架构中通用它的直译, 也就是“界面”, 而不是“接口/编程接口”, 因为有些架构中的界面, 并不是用编程来实现的。在这种情况下, 界面更多的是一种规约。



1



晓小东

2019-12-27

老师对于下面两段话, 我理解的不是很清楚 (没看出来, 判断两次还是判断一次逻辑???)

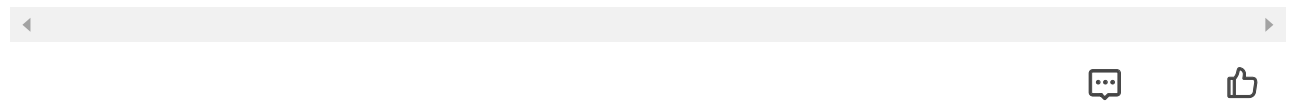
NOTE: 在 ECMAScript 6 之前, 由于[PrimitiveValue]来存放对应的封装类。也就是说, 只有当obj.[Class]存放着false值时, 它才是false值所对应的对象实例。而 ECMAScript 6

将上述的依赖项变成了一个，也就是说只要有一个对象有内部槽[[BooleanData]]，那么...
展开 ▾

作者回复: ES6之前，是需要判断两次的。

- * 有[[PrimitiveValue]]内部槽，说明是一个用包装类得到的值。然后，
- * 查看[[Class]]内部槽，找到对应的包装类，从而知道类型。

在ES6之后，由于每种包装类有独立的一个槽，所以如果对象obj有[[BooleanData]]，那就说明了包装类是Boolean()，且被包装的数据在[[BooleanData]]槽中。



王大可

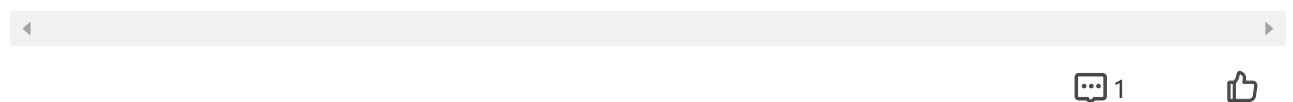
2019-12-26

在chrome浏览器（版本 79.0.3945.88（正式版本）（64 位））计算 {} + {} 结果是"[object Object][object Object]"
edge 下是计算 {} + {} 结果是 NaN

作者回复: 确实。

不过在浏览器的控制台上，和在引擎的层面上执行也是会有区别的。都是使用 v8，NodeJS在Shell中与chrome也一样。但是你写在.js文件中，或者直接从node的命令行上执行，效果就不一样了，例如：

```
...  
> node -p -e '{} + {}'  
NaN  
...
```

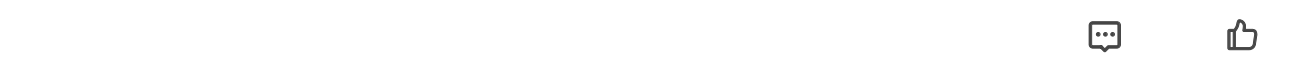


鲜于.css

2019-12-26

js就是学不明白闭包和原型原型链

展开 ▾



晓小东

2019-12-25

老师有个问题，既然您讲了数据的值类型与引用类型概念，像weakSet与weakMap 对对象的弱引用该如何理解，这个弱引用到底是个啥。

展开 ▾

作者回复: 弱引用是向weakSet/weakMap中添加一个目标对象的引用，但添加是目标对象的引用计数不增加。比较来说：

```
...  
var x = {}; // <-右边的对象字面量的引用计数加1  
var y = x; // <- 再加1  
weakSet.add(x); // <-不加1  
weakSet.add(y); // <-也不加1  
delete x; // 减1  
delete y; // 再减1  
...  
...
```

到这里，由于对象的引用计数为0了，所以weakSet中的那个被add()进去的x、y就自动被回收了。——weakSet/weakMap具备这种机制。

所以weakSet/weakMap没有size这个属性，它不安全。——你刚读了它的值，它自己自动回收了一下，就又变掉了。

